Introduction to Multimedia Content Analysis

Multimedia content analysis tries to infer from the content of multimedia, i.e. from images, audio, or video with the goal of enable interesting applications. For example, finding all the smiling faces in a video to extract portrait pictures of them is an application of multimedia content analysis. Another example is search based on keywords spotted in the audiotrack of a movie. Later chapters will explore some of the applications of multimedia content analysis, such as information retrieval further. In this chapter we will provide a short introduction to the basics of machine learning that are vital to understand research work in multimedia content analysis.

Historically, even though the mathematical foundations are very similar, until recently multimedia content analysis research seemed to be strictly divided according to the type of data that were to be analyzed. Therefore many researchers work on either acoustic processing, computer vision, or natural language processing. Only recently multimodal content analysis has emerged trying to merge the different types of sensory data to create unified *multi*media approaches that can benefit from the synergy of leveraging modalities in a combined way.

The foundations of audio, speech, image, and video content analysis are rooted both in the signal processing community, which is part of electrical engineering, as well as the field of statistics, which is part of mathematics. Many terms that are still used have been inherited by these two fields. A newer field, which has come up with the raise of the computer, is called machine learning. It is a subfield of statistics, paired with some biology and neuroscience roots. The field of machine learning redefines many terms originally created in statistics and biology. The application of machine learning together with the foundations of signal processing to different kinds of data created the different fields of audio, speech, image, and video processing, which by themselves created new terms. There are different reasons for those fields to have become separated, some are social and very pragmatic. A very important one though is the amount of data that has to be processed. Audio and speech processing is the oldest field because computers where already able to process speech in the 60s and 70s. Image analysis is a slightly newer field, and video analysis is the newest because there is much more data to be processed. With different maturities of the fields, different generations of people have worked on the different types of data and hence, different vocabulary is used.

Today, the processing capabilities of modern computers allows us to think about approaches that analyze multimedia multimodaly, i.e. processing audio, images, motion, and meta-data synergistically. A combined processing promises improved robustness in many situations and is closer to what humans are doing: The human brain takes into account not only patterns of illumination on the retina or periods of excitation in the cochlea, it also combines different sensory information and benefits from past experience. Humans are able to use context information and to fill in missing data by associating parts of objects with already learned ones.

The following chapter provides a short introduction to the field of multimedia content analysis before the next chapters dig deeper into the specialties of individual sensory modalities.

Basics of Machine Learning

The basic workflow of a machine learning approach is illustrated in Figure 1.

Using training data, relevant features are extracted and passed to a machine learning algorithm. These algorithms, are usually methods derived from mathematical statistics and produce statistical models. These are basically sparse representations of the data that allow thresholding of any kind. To train the models, the right answers have to be provided which are given in the ground truth data. This is usually metadata created by human annotators. Currently Common algorithms include Gaussian Mixture Models, Neural Networks, Support-Vector Machines,

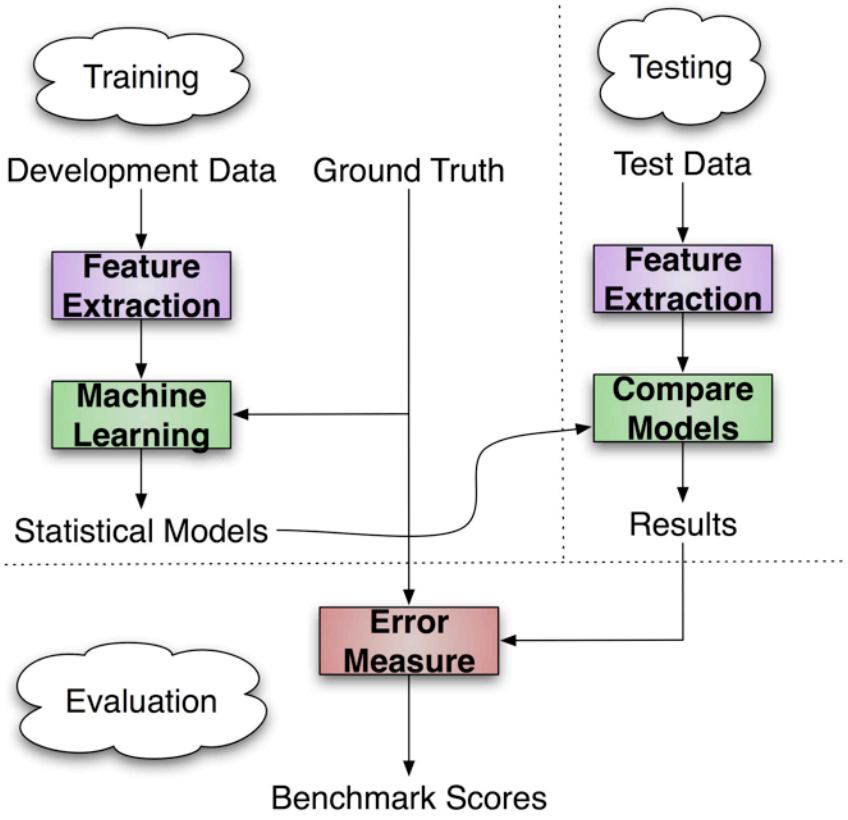

Figure 1. Typical workflow of content analysis

Decision trees (for example ID3), k-Nearest Neighbors, Bayesian Networks, and Hidden Markov Models. We will describe the basic ideas of these algorithms below. Feature extraction methods are usually derived from signal processing or electrical engineering, we will describe common features later.

In testing mode, the statistical models are then used to perform the actual content analysis task. The test data is run through the same feature extraction process. The results are either just used for the actual application or compared against the ground truth to benchmark the quality of the algorithm.

Here is an example: Let's assume we want to detect the gender of a speaker by their voice. The development data consists of samples of female and male speech from humans of different ages

uttering different text. From the raw audio files, features are extracted that try to reduce the amount of raw data while emphasizing speaker characteristics (features are further explained in Chapter XXX). These are then used to train a statistical model for the two classes. In testing mode, a random audio snippet containing speech and undergoing the same feature extraction is then presented to the model for classification. The output may be used for anything. In evaluation mode, however, the quality of the classification is measured by comparing the classification results with the ground truth for many test cases and counting both the number of right and wrong classifications. In our example, an error might be expressed as percentage of the wrongly classified test samples versus the total number of test samples.

Unsupervised machine learning, as opposed to supervised learning, omits the training step and statistical models are created on the fly using the test data. Evaluation is performed in a similar way.

Supervized Modelling

In the following, we will discuss some common supervized learning techniques often used in multimedia. Supervized learning uses a development set to learn a model for the mapping of sample values to classes (classification task) or a model for sample values to a continuous range (regression task). A typical classification task is the answer to the question “does this portion of audio contain speech?” or “which digit can be seen in the picture?”, a typical regression task is “what are the geo-coordinates of the position of the camera in this image?”. The list of presented algorithms here and the explanations are not exhaustive. As explained above, machine learning is it’s own field and interested you are invited to explore the literature for further reading.

k-Nearest Neighbors

The most basic learning technique, which can almost not be called learning technique is nearest neighbor search. Let’s assume one has a dataset with a number of samples in an N-dimensional space e.g., 8x8 binarized scanned digits (compare also first example in Chapter XXX). Each sample is labelled, i.e. we know to which class it belongs. A nearest neighbor search assumes that the dataset spans an N-dimensional space. This space is our ``trained” knowledge from the development data.

Now, given a new sample without label, i.e. from the test data, we search the development data for samples with low distances from the sample. We then take the labels of the k samples with lowest distances and vote: The label that is in the majority wins. Obviously, with *k=1* the decision is always clear and also it is usually be a good idea to choose an odd number for *k* even though, depending on the number of classes, this will not always guarantee an inambigous decision. A rule of thumb is that *k=1*, *k=3* or *k=5* are good numbers. If these often result in two many different candidate labels to make a decision, either the distance metric or the k-nearest neighbor algorithm is the wrong choice. Nearest neighbor search with *k=1* was also referred to as the post-office problem, referring to an application of assigning a residence to the nearest post office. See literature.

The following pseudo code describes a k-nearest neighbor search with an arbitrary distance metric.

```
// Input: A number N of labelled samples S with label set L,
// k number of neighbors to take into account,
// f distance function,
// x, a sample to be classified
// Output: A label for x
kNN(S, L, N, k, f, x)
    D := [] // array of distances with length = k
    I := [] // array of indizes to neighboring samples with length k
    C := [] // array of counts for a label
    FORALL d IN D  // initialize all distances to infinity
          d := +infinity
    FORALL c IN C // initialize counts to zero
          c:=0
    FOR i:=0 to N
          distance := f(x,X[i])
          FOR j:=0 TO k-1
                IF (distance<D[j])
                      D[j]:=distance
                      I[j]:= i
    FOR i:=0 TO k-1
          C[L[I[i]]]++ // increase count for label
    Y:=max_index(C) // Take the label with the maximum count,
                    // note: decision might be ambiguous
kNN := Y
```

Obviously, the quality of the classification results will depend on many factors, including the chosen distance metric which determines when two samples are considered similar. Also, it can been shown that under a broad set of conditions, as dimensionality increases, the distance to the nearest data point approaches the distance to the farthest data point. Therefore, higher dimensional data usually requires different techniques. More importantly, the algorithm does not create a generalized model for the classes, i.e. a small cluster of outliers can decrease the quality of the classification result. Also, the kNN algorithm requires comparing the test sample with each sample in the training set, which can be quite expensive. These thoughts lead us to the next algorithm.

k-Means and k-Gaussians

We already presented the k-Means algorithm in Chapter XXX (compression) as a vector quantization algorithm. By keeping the labels intact, k-Means can be a good solution to reducing the size of the development set for kNN as comparing to the means instead of the whole data will reduce the number of comparisons. Also, the means can be seen as abstract representation of the data in that they represent more than one individual value and therefore make the comparison less prone to classification errors due to outliers. k can be chosen to match the number of classes in the development set or increased for finer granularity.

When using the means to model data, it is often desired to have an indication of the variance of the represented cluster as well i.e. how close to the mean is the typical sample in the cluster. Also, as can be observed in nature, often cluster values have the typical bell-curve shape. Therefore, a very frequent way of modeling data beyond k-Means is modeling it with *k* Gaussians. A function *f* for a Gaussian is defined as

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}},$$

with μ being the mean and σ^2 the variance. Figure 2 shows the typical bell-shaped curves for different means and variances.

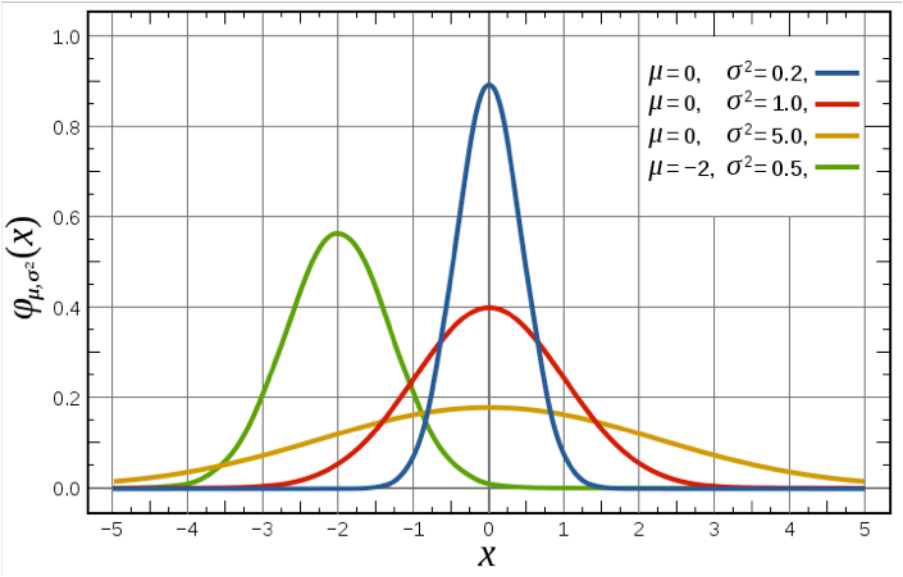

Figure 2. A set of Gaussians with different μ and σ^2.

Modeling data with k Gaussians only requires a slight modification of the k-Means algorithm: After the means have been determined, the variance σ^2 has to be calculated. This can be done by interpreting the sample values as observations as defined by probability theory. The Gaussian function becomes a probability density function:

$$f(x;\, \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-(x-\mu)^2/(2\sigma^2)}, \quad x \in \mathbb{R}.$$

and the set of samples are interpreted as values of a discrete random variable X. The variance

$$\sigma^2 = \sum_{i=1}^{n} p_i \cdot (x_i - \mu)^2$$

with probability mass function $x_1 \mapsto p_1$, ..., $xn \mapsto pn$,

In order to classify a test sample, the probability of the sample belonging to each of the k Gaussians is calculated and the Gaussian with the highest probability is chosen. The pseudo code for calculating k-Gaussians in analogy to k-Means is left as an exercise.

Mixture Models

More than often, clusters do not exactly adhere the Gaussian function. For example when there are two or more "bumps", i.e. the distribution is multimodal. This is especially the case when clusters are the mixture of two or more independent components. For example, a color model for face images might consist of the mixture of a model for the skin, a model for hair, a model for lips, and one for eyes. Obviously, different components will have different importance values, which can be expressed by weighting the individual components. In general, a probability mixture model f_X is defined as a weighted sum of its component distributions:

$$f_X(x) = \sum_{i=1}^{n} a_i f_{Y_i}(x)$$

with X being a discrete random variable and a mixture of the n-component discrete random variables Y_i for some mixture proportions $0 \leq a_i \leq 1$ where $a_1 + \cdots + a_n = 1$.

Like k-Gaussians and k-Means, a Gaussian Mixture Model can also be learned by Expectation Maximization (EM). As explained earlier, EM is an iterative method for finding most likely parameters for models (when probabilistic models are used, the likelihood is maximized). The algorithms alternates between the so-called expectation step (E-Step) which estimates the expected likelihood given the current model, and a maximization step (M-Step), which computes new parameters given the current memberships, maximizing the expected log-likelihood found on the E step. These new parameter-estimates are then used to determine the distribution of the memberships in the next E step. This is repeated until the system converges based on an evaluation metric or a fixed amount of iterations. The mixing coefficients a_i are the means of the membership values over the N data points:

$$a_i = \frac{1}{N} \sum_{j=1}^{N} y_{i,j}$$

The model parameters f_{Y_i} are calculated by using the data points x_j that have been weighted using the membership values. For example, for a Gaussian, μ is calculated using:

$$\mu_i = \frac{\sum_j y_{i,j} x_j}{\sum_j y_{i,j}}.$$

The implementation of Gaussian Mixture Model training is left as an exercise. The concept of Gaussian Mixture Models is rather straightforward, at the same time, Gaussian Mixture Models are a very powerful concept used in many places in multimedia research, especially in visual and acoustic object recognition (e.g., in interactive image segmentation and speaker identification). In practical approaches, the number of Gaussians per cluster, the number of clusters (unsupervized case), and the number of training iterations and/or the convergence criterium are important factors that have to be empirically determined. Using too few Gaussians may make the likelihood estimation poor and therefore not work well for discriminating between the different clusters on the test data. Using an unlimited number of clusters or Gaussians in a mixture model allows to model an arbitrary distribution. However, using too many Gaussians results in so-called

overfitting of the data, for example when each data point is represented by its own Gaussian, the models are too specialized and not useful for representing the unknown test data. In other words, there is a trade-off between the number of parameters that should be used to describe the model given the number of training samples and the complexity of the distribution: To few parameters result in a bad model, too many parameters result in a bad abstraction. Chapter XXX already describes the Bayesian Information Criterion, which is one way of objectively approaching the trade-off.

Artificial Neural Networks

An Arificial Neural Network (ANN), under computer scientists often simply called "Neural Network" (NN), is another statistical modeling technique. The name stems from its motivational background of trying to simulate some of the structure and/or functional aspects of biological neural networks. ANNs usually consist of an interconnected group of components (often called artificial neurons) of which each of them simulates one function. The artificial neurons, i.e. the functions, are connected in layers with each group of neuron feeding its output to a corresponding neuron in the next layer. In the end, the goal is to model a function f:X->Y, with X being the input data and Y typically being a desired regression or classification output. In most cases, an ANN is an adaptive system that learns its structure from the training data by assigning weights to the connections and parameters of the neuron functions. The most common type of ANN used in practice is the so-called Multi-Layer Perceptron (MLP). A Multi-Layer Perception is an ANN composed of many interconnected Perceptrons.

A Perceptron is a basically treshold function which maps its input x (a real-valued vector) to an output value $f(x)$ (a single binary value) across the matrix.

$$f(x) = \begin{cases} 1 & \text{if } w \cdot x + b > 0 \\ 0 & \text{else} \end{cases}$$

where w is a vector of real-valued weights and is the dot product (which computes a weighted sum), b is the so-called bias, a constant term that does not depend on any input value. So the value of $f(x)$ (0 or 1) is used to classify x as either a positive or a negative instance. If b is negative, then the weighted combination of inputs must produce a positive value greater than $|b|$ in order to push the classifier neuron over the 0 threshold. Perceptrons take multiple inputs x_i by summing them together:

$$f(x) = f(w_0x_1 + w_1x_1 + w_2x_2 + \ldots + wmxm) + b)$$

As can be seen, a single Perceptron therefore can divide the input space into two areas, for which one area is classified as 1 and the other as 0. The division is a line in two dimension, an area in 3 dimensions, and a hyperplane in in any higher dimensionality. The Perceptron is therefore said to be *linearly separating* the space. The following algorithm can be used to to train the parameters w and b given a set of input points and a set of classification labels for the input points (training data).

```
// Input: Training data with n samples x[i] and desired output labels l[i]
// Output: Perceptron parameters w[i] and b
p_learn(x[i],l[i], n)
    Initialize the w[i] and b randomly by
    setting w[i][t] (0<=i<=n) to be the weight at iteration t, and
    b to be the bias in the output node, and
    w[0]:= -b, and x[0]:=1, and
    settting w[i][0] to small random values.
    t := 0
    DO
        FOREACH sample x[i]
            // calculate output
            output[t]:=
                (w[0][t]*x[0][t]+w[1][t]*x[1][t]+...+w[n][t]*x[n][t])+b
            w[i][t+1]:= w[i][t]+gamma*(l[i] - output[t])*x[i][t]
            // where 0<=gamma<=1 is a value to slow down the adaption rate
        t := t+1
    UNTIL t>threshold
p_learn := (w[i],b)
```

Practically, two problems have to be addressed for Perceptrons to be applicable to everyday classification problems. First, the Perceptrons have to be able to also separate spaces that are not linearly separable but require more complicated function. Second, the weights and biases should be tuned automatically based on a training set. The following paragraphs will shortly explain how to do both, however the theory of Artificial Neural Networks is an ongoing field of research. and many other networks exist in addition to the basic ones presented here. See literature for more details.

In order to be able to separate input space into more than two classes and also to separate a problem space using more than linear separation several Perceptrons are connected allowing piece-wise linear approximation of the separation function. A so-called Multilayer Perceptron (MLP) is a feedforward artificial neural network model that maps sets of input data onto a set of appropriate output. It is a modification of the standard linear Perceptron in that it uses three or more layers of nodes (so-called neurons) that are interconnected, i.e. the output of the neurons in one layer feeds the inputs of the neurons in the next layer. The typical structure looks like the one sketched in Figure 3.
The input layer is the layer of nodes receiving the input directly. The output layer is the layer that outputs the functions directly. The so-called hidden layer is the layer in between. One can show that an MLP with three layers can approximate any function given enough neurons in the input, output, and hidden layers. Figure 3 shows an example of an MLP calculating the binary XOR function. For solving multimedia classification problems, many more hidden nodes are needed. A practical rule of thumb is that the number of total parameters in a Neural Network should be about 1 per 10-20 parameters to be learned.

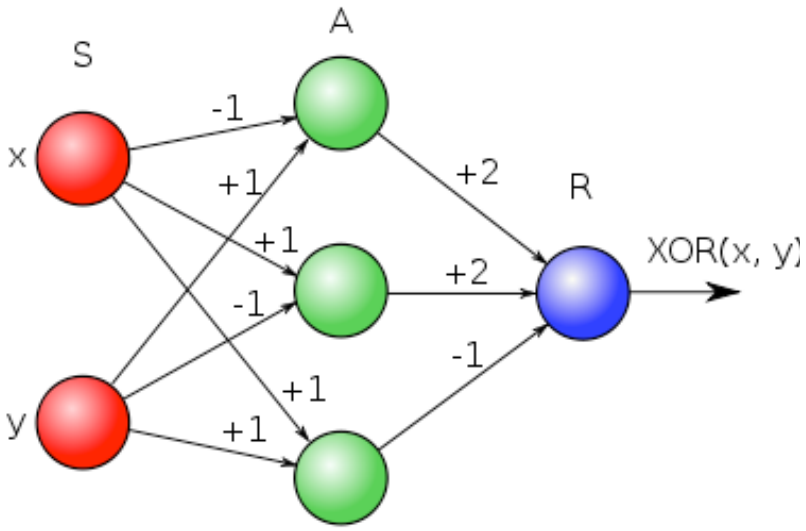

Figure 3. A Multilayer Perceptron (MLP) for calculating the binary function XOR. The numbers on the edges define the weights *w*. The bias *b=0*.

While the network in Figure 3 is still straightforward to design manually, MLPs with hundreds or thousands of hidden layers are not. Therefore, different ways of learning the weights based on training data. The most well-known algorithm is called backpropagation. The idea is that the weights are learned by calculating the output given the current weights and then calculating the difference between the output and the training data. When doing this for many samples, one can extrapolate an error function given the weights and then tune the weights following the gradient of that error function to a minimum. To be able to calculate the gradient both the error function and the Perceptron function must be differentiable. Therefore, mean-square error is often used to compare output and training data and the Perceptron function is often replaced with the so-called soft-max function, which is defined as:

$$p_i = \frac{\exp(q_i)}{\sum_{j=1}^{n} \exp(q_j)},$$

with p_i being the values of an output node, $q_{i,j}$ are the net inputs to the output nodes, and n is the number of output nodes. It ensures all of the output values p are between 0 and 1, and that their sum is 1. This also makes it easier to interpret the outputs of a node as probabilities.

The following constitutes pseudo-code for the back propagation algorithm:

```
// Input: An MLP network with hidden and input weights wh and wi,
// Training data T
// Output: A network with new weights
backprop(network(w[i]),T)
    Initialize the weights in the network randomly
    DO
          FOREACH sample e in T
            output := compute_output(network,e) // forward pass
            mse := mean-square error (T[e] - output) at the output layer
            gradient := mse * e
            FOREACH wh in the hidden layer:
                Compute delta(wh) // backward pass
```

```
                wh := wh - gradient
            FOREACH wi in the input layer:
                Compute delta(wi) //backward pass
                wi := wi - gradient
    UNTIL mse<threshold
backprop := network
```

Support Vector Machines

Another approach to solve the supervized classification problem is the so-called Support Vector Machine (SVM). Like Neuronal Networks this is more a class of algorithms rather than a single one and they are constantly evolving. Therefore the reader is, again, strongly encouraged to follow the most recent literature on this field of specialization. Like the Multilayer Perceptron, a SVM performs classification by constructing an *N*-dimensional hyperplane that optimally separates the data into two categories:

$$f(x) = \begin{cases} 1 & \text{if } w \cdot x + b > 0 \\ 0 & \text{else} \end{cases}$$

Not only is the above equation familiar, SVMs are closely related to neural networks: An SVM using a sigmoid kernel function is equivalent to a two-layer MLP. In contrast to MLPs, SVMs regularly solve only two-class problems. The following paragraphs explain the main concepts.

The goal when using an SVM is to find the optimal separation with a hyperplane that discriminates two clusters of input data in such a way that the sample within one category of the target outcome are on one side of the plane and cases with the other category are on the other size of the hyperplane. The vectors closest to the hyperplane are called the *support vectors* as they define the separation margin: SVM analysis finds the hyperplane that is oriented so that the margin between the support vectors is maximized. Figure 4 illustrates the idea based on a two-dimensional example.

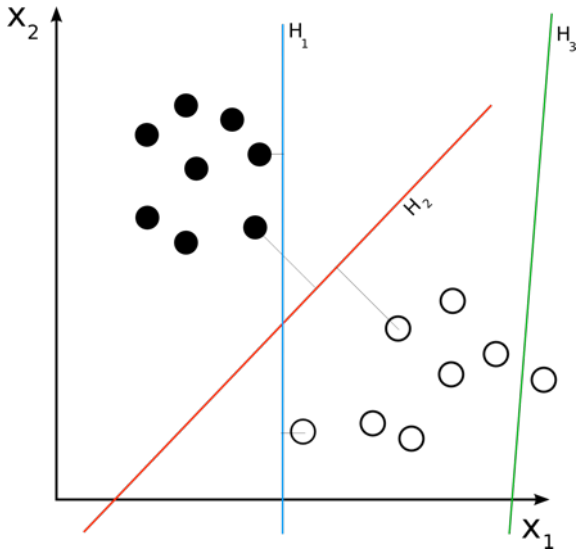

Figure 4. Different (hyper-)planes in space. H_3 does not separate the space correctly. H_2 and H_1 do, but only H_2 maximizes the margin between the two classes (and the support vectors).

A simple two-dimensional case can be solved using dynamic programming or the Perceptron learning algorithm (see above). However, as explained earlier, often data is not linearly separable. While MLPs implicitly learn to fold the input space when training the hidden layer of the network, SVMs use a different trick: They use a so-called kernel function to manipulate the input values so that they may be separated by a hyperplane. In other words: Rather than seeking for a non-linear separation function, they transform the value space to be linearly separable. A major trick is to increase the dimensionality of the input space. This trick *always* allows to find a linear separation if the dimensionality is chosen high enough. Figure 5 illustrates the idea.

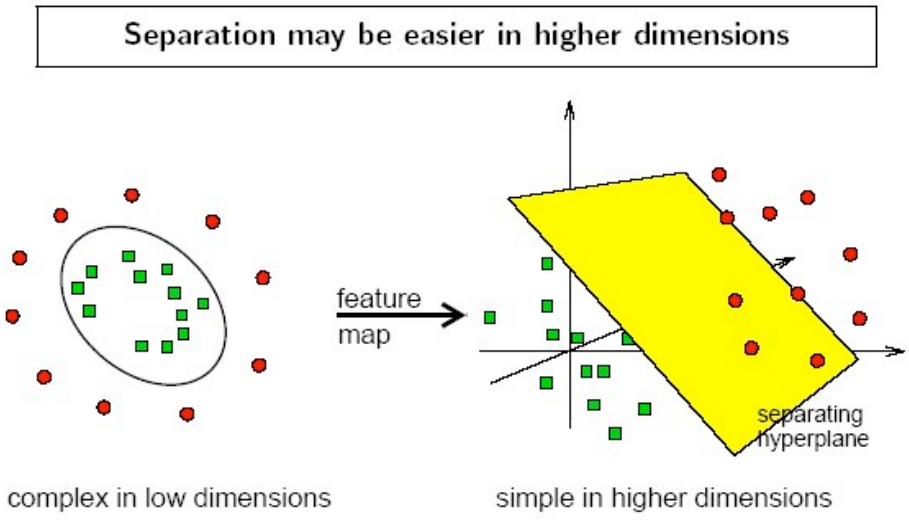

Figure 5. Separation of the input space is always easier in higher dimensions. TODO: REDO IMAGE. (Copyright status not clear)

Mathematically, a kernel can transform any algorithm that solely depends on the dot product between two vectors. Wherever a dot product is used, it is replaced with the kernel function. Thus, a linear algorithm can easily be transformed into a non-linear algorithm. In practice, many kernel mapping functions may be used and finding the right one can be tricky. However, a few kernel functions have been found to work well in for a wide variety of applications and seem to predominant in scientific literature. The most simple of them is the linear kernel:

$$K(x, x_i) = x_i^T x,$$

with x_i being the input values. The two most frequently used one are probably the polynomial kernel

$$K(x, x_i) = \left(1 + x_i^T x/c\right)^d,$$

and the radial basis function (RBF) kernel

$$K(x, x_i) = \exp\left(-\left\|x - x_i\right\|^2 / \sigma^2\right),$$

where d, c, and σ are constants. If one chooses a sigmoid function as kernel, such as

$$K(x, x_i) = \tanh\left(k\, x_i^T x + \theta\right),$$

with k and θ being constants again, one can emulate an MLP (as outlined above).

Finally, to allow for some flexibility in separating the categories, SVM models can have a cost parameter C, that controls the trade off between allowing training errors (e.g. allow some points to be wrongly classified) and forcing rigid margins. This is called *soft margin classification*. Increasing the value of C increases the cost of misclassifying points and forces the creation of a more accurate model that may not generalize well.

What we need now is an algorithm to train an SVM. In general, all it involves is solving the following optimization problem. As explained so far, a general SVM can be expressed as

$$u = \sum_i \alpha_i y_i K(\vec{x}_i, \vec{x}) - b$$

where u is the output of the SVM, K is the kernel function which measures the similarity of a stored training example x_i to the input x, $y_i \in \{-1,1\}$ (given by the training labels) is the desired output of the classifier, b is a threshold or bias, and a_i are weights which blend the different kernels. Training therefore consists of finding the weights a_i, which is usually expressed as a minimization of the following formula:

$$\min_{\vec{\alpha}} \Psi(\alpha) = \min_{\vec{\alpha}} \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} y_i y_j K(\vec{x}_i, \vec{x}_j) \alpha_i \alpha_j - \sum_{i=1}^{N} \alpha_i,$$

subject to constraining the range of the weights and the constraint that

$$\sum_{i=1}^{N} y_i \alpha_i = 0.$$

The most common algorithm to solve the problem is the so-called Sequential Minimal Optimization (SMO) algorithm, invented by John C. Platt (see references). The main idea behind the algorithm is, again, to break up the large problem of optimizing the margin into a number of smallest possible problems, thereby achieving a O(n log n) runtime. The algorithm is described in pseudo-code in the original reference (XXX).

Markov Chains

The machine learning algorithms described so far are all trying to classify or model particular patterns or samples of data independently of each other. Especially when handling multimedia data, however, it is very often that case that a particular sample depends on the previous sample. For example, when parsing English language in the form of text or speech the probability of a vowel followinga 'y' is much higher than the probability of a 'j' following a 'y'. These kind of temporal dependencies can be modeled in several ways, including with ANNs. However, the most common way in literature is the the use of a Markov chain, and more particular the use of a Hidden Markov Model (HMM).

A Markov chain is a stochastic model that assumes the Markov property, which is that if the conditional probability distribution of future states of the process depend only upon the present state. In other words, given the present, the future does not depend on the past. Generally, this assumption enables reasoning and computation with the model that would otherwise be computationally intractable. A Hidden Markov Model (HMM) is a Markov chain for which the state is only partially observable, i.e., observations are related to the overall state of the system, but they are typically insufficient to precisely determine the state. A typical problem is that one is given a sequence of observations and has to compute most-likely corresponding sequence of states. Figure 6 shows a very common example from literature.

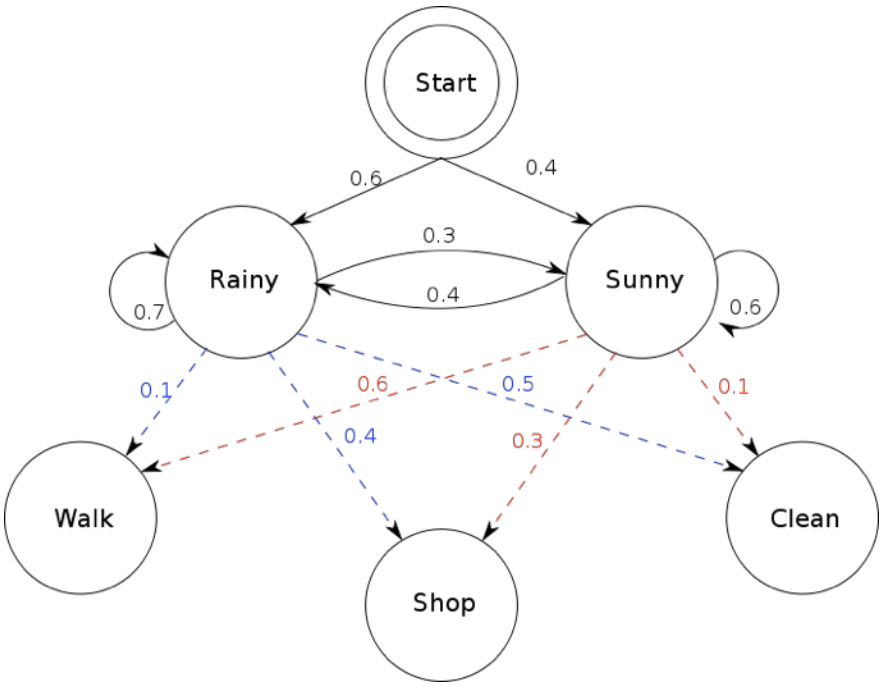

Figure 6. An example HMM modeling activities given weather probabilities.

Assume two people living far apart from each other but talking over the telephone frequently about what they did each day. One of them is only interested in three activities: walking on the beach, shopping in a mall, and cleaning her own apartment. The choice of what to do is determined exclusively by the weather on a given day. The other person on the phone has no definite information about the weather and also doesn't dear to ask but knows general trends. Therefore, based on the activity reported each day, the other person tries to guess what the weather must have been like.

To do this, the weather and activities are modeled in a Markov chain as depicted in Figure 4. There are two states for the weather, "Rainy" and "Sunny", which as pointed out in the description cannot be observed directly. In other words, they are hidden. However, on each day, there is a certain chance that the person on the phone will perform one of the following activities: "walk", "shop", or "clean", which, as we know, depend on the weather states. Since we know about the activities, we call these states observations. Based on the general trend of weather (climate) and the statistics of the average type of activity, one can label the edges of the HMM with probabilities.

In this model, the probabilities emitted by the start node represent the belief about which weather state of the HMM is in based on climate. The probability edges between the weather nodes represent the transition probability of the weather, in other words how likely is the weather to stay the same or change. In our example, there is only a 30% chance that tomorrow will be sunny if today is rainy. Finally, the so-called emission probability represents how likely it is that a certain activity is performed each day given the weather. In this model, there is a 50% chance of the cleaning apartment activity when it is rainy and a 60% chance for the walking state when it is sunny.

In order to understand how a HMM helps with temporal modeling let us consider this example: Out two phone chatter talk to each other three days in a row. The activity sequence looks like this: walk-shopping-cleaning. The question now is: What is the most likely sequence of rainy/ sunny days that would explain these observations?

This is very frequent problem, especially when modeling language. An analog for speech recognition would be: Given the observation of the likelihoods for certain phones, what is the probability of the speaker having uttered word x?

Given an HMM, the answer to this problem is given by a very popular algorithm, named after his creator Andrew Viterbi, the so-called Viterbi algorithm. The idea behind the algorithm is that builds a graph of all possible state transitions. Then, the path with the maximum probability is chosen, the so-called Viterbi path. The following pseudo code shows the algorithm:

```
// Input: A sequence of observations obs[], hidden states hstates[],
// start_p is the start probabilty, transp[] and emit_p[] the transition
// and emission probabilities, respectively.
// Output: The output probability of the most likely path.
viterbi(obs, hstates, start_p, trans_p, emit_p)
  graph = []
  path = []

  // initialize
  FOR y IN hstates
      graph[0][y] = start_p[y] * emit_p[y][obs[0]]

  FOR t:=1 TO LENGTH(obs)
      newpath = []
      FOREACH y IN hstates
           prob := max([(V[t-1][y0] * trans_p[y0][y] * emit_p[y][obs[t]])
                   over all y0 in hstates
      graph[t][y] = prob
   prob := max((V[LENGTH(obs) - 1][y]) over all y in hstates
viterbi := prob
```

Decision Trees

A decision tree is a decision support tool that uses a tree-like graph structure to model decisions and their possible consequences, sometimes including the probabilities of event outcomes, resource costs, and other factors. Figure 7 is a photograph taken from Wikipedia showing a whiteboard with a typical decision tree. A decision tree is a widely-used tool in many areas. In multimedia, decision trees are particularly useful to combine different media. For example, different sensors might output different values for different input (e.g. audio classification determines car noise, video classification determines a street -- what is the probability for a highway scene?). In order to combine the sensors, a decision tree might help determine the final outcome based on the different measurements. Often, the trees are created manually. However, learning them automatically based on training data is often desirable because the tree might become rather complex. Again, the topic of decision tree learning is an ongoing field of research

and we cannot present it exhaustively in this book. Therefore, we limit ourselves to one of the most used algorithms, called C4.5.

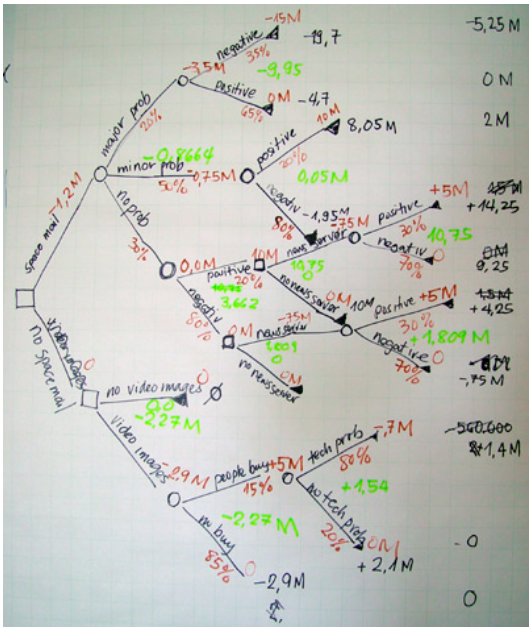

Figure 7. A manually created decision tree on a whiteboard.

C4.5 has been created by Ross Qinlan and can be used for classification. It is used very frequently in many multimedia projects, mostly for analyzing the combination of textual features. The idea of the algorithm is straightforward: The tree is build iteratively and at each node feature ranges are split so that they most effectively separate its set of samples into subsets enriched in one class or the other. In order to find which subset would "enrich" a class more, C4.5 defines the notion of information gain, which in turn is based on Entropy. The information gain over a split is:

$$G(S,A) = E(S) - \sum_{i=1}^{m} f_S(A_i) E(S_{A_i})$$

where $G(S,A)$ is the gain of the subset S after a split over the A features, $E(S)$ is the information entropy of the subset S, m is the number of different values of the features A in S, $fS(A_i)$ is the frequency of the items possessing Ai as value for A in S, A_i is ith possible value of A, S_{Ai} is a subset of S containing all items where the value of A is A_i. $E(S)$ is defined as follows:

$$E(S) = -\sum_{j=1}^{n} f_S(j) \log_2 f_S(j)$$

with n is the number of different values of the features in S (entropy is computed for one chosen feature), $fS(j)$ is the frequency of the value j in the subset S. The following pseudo code demonstrates the idea:

```
// Input: A sequence of observed examples[], a sequence of attibutes[],
// a target attribute ta.
```

```
// Output: A labelled decision tree T with children labelled +/-
c45(examples,attributes,ta)
    T := root_node
    IF all e in examples > 0:
          root_node.label := "+"
          return T
    IF all e in examples < 0:
          root_node.label := "-"
          return T
    IF number of predicting attributes is empty:
      return the single node tree Root, with label := most common value
      of the target attribute in the examples
    FOREACH attribute a
          Find the information gain from splitting on a
    A := The attribute that has highest information gain
    Decision tree attribute for root = A.
          FOREACH a of A:
                Add a new tree branch below root,
                corresponding to the test A = a.
                Let examples(a), be the subset of examples
                that have the value a for A
                IF examples(a) is empty
                      Then add a leaf node with label :=
                      most common target value in the examples
                ELSE add the subtree C45(Examples(a), attributes - {A}, ta)
c45 := T
```

Since trees can become very complex, the C4.5 algorithm defined a second routine that iterates through the tree once it's been created and attempts to remove branches that do not help by replacing them with leaf nodes. Sometimes, features can be associated with a cost function to return different information gains depending on their use. This may be especially interesting when combining different sources of sensor data, since they may have different validity.

Unsupervized Modelling

All the methods explained so far assumed the presence of training data. Sometimes, however, training data is not available or it is not practical to ask users to provide it. Still, it might be desirable to label multimedia data according to their similarity. This is called clustering and examples of clustering were already explained in Chapter XXX (lossy compression): The k-Means and X-Means algorithms are perfectly suitable for these tasks.

Often, however, because of the type of data one is dealing with it is desirable to use the same machine learning algorithm one would use in the supervized case, for example, when certain machine learning methods have proven to work well with a certain type of feature. Fortunately, all one has to do is to replace the error function, which is based on comparison with ground truth training data in the supervized case, with a similarity function for the unsupervized case. The following pseudo-code illustrates Gaussian Mixture Model training (as discussed earlier) for the unsupervized case:

```
// Input: A sequence of n samples x[], a number of clusters k
// Needs: k-Means, Gaussian function f(x,mu,sigma), sum()
```

```
// Output: k parameter triples (a[],mu[],Sigma[]) that describe k
// weighted Gaussians
GMM_train(x[],k)
  Initialize a[],mu[],Sigma[] by using k-Means clustering
  Initialize probabilities p[][]:=0
  iter:=0;


  DO
    // E-Step
    Compute probabilities p[i][j]:=(a[j]*f(x[i],mu[j],Sigma[j]))/
                                   (sum(a[j]*f(x[i],mu[k],Sigma[k])))
              for all elements in x[i:=0..n] and all clusters j:=0..k.
    // Needs to be stored in an n×k matrix. Each individual cell of this
    // matrix corresponds to probability that x_i belongs to the Gaussian
    // distribution specified by mu[k],Sigma[k].

    // M-Step
          Update the weights a[j]:=sum(p[i][j]/n) for j:=0..k
          Update the centroids mu[j]:=sum(x[i],p[i][j])/sum(p[i][j])
          for j:=0..k
          Update the co-variances
          Sigma[j]:=sum(p[i][j]*(x[i]-mu[j])(x[i]-mu[j]))/sum(p[i[[j])
          for j:=0..k
    iter++;
  WHILE (iter<threshold)
GMM_train := (a[],mu[],Sigma[])
```

A typical general pattern for unsupervized training is bottom-up and top-down agglomerative hierarchical clustering. One way to think of the concepts is to think of them as sorting algorithms: A divisive approach can be implemented recursively like quicksort. An agglomerative approach is comparable to bottom-up mergesort. So in general, for divisive segmentation/clustering we start at the top with all frames in one cluster. The cluster is split using a flat segmentation algorithm. This procedure is applied recursively until the algorithm determines no further splitting is necessary or each frame is in its own singleton cluster. Top-down clustering is conceptually more complex than bottom-up clustering since we need a second, flat clustering algorithm as a ``subroutine''. However, it has the advantage of being more efficient if we do not generate a complete hierarchy all the way down to individual document leaves. Different algorithms are used in different situations. So again, we suggest further study of machine learning literature.

The following pseudo code exemplifies bottom-up clustering for a generic similarity metric:

```
// Input: A sequence of samples[] (e.g. audio),
// a minimum number of frames for each model segmentlength,
// a similarity metric as a function,
// a treshold
// Output: A sequence labels[]
divisiveclustering(samples, segmentlength,metric(),threshold)
    IF #samples<windowlength:
          return
    run metric() over segments of samples of segmentlength
    determine maximum max for all windows
```

```
    IF max>threshold:
          split at sample s with metric() result m
          run divisiveclustering() for samples before s
          run divisiveclustering() for samples after s
          FOREACH segment seg:
                Using metric() compare seg with other segment
                IF results<threshold:
                      give same label to seg and other segment
    ELSE:
          return
divisiveclustering:=labels
```

The reader ask what a suitable implementation for metric is. Again, this depends on the nature of the problem and the underlying data. If an obvious measure cannot be found analyticly (which is the case for most practical multimedia problems), only empirical measurement can quantify the quality of an approach (see also next section on error measurements). Nevertheless, some metric seem to have predominant status in the research community, such as the Minkowski metric and especially the two special cases the Manhattan Distance and the Euclidean Distance.

The Euclidean Distance *d* of two points in n-dimensional space *p* and *q* is defined as:

$$\mathrm{d}(\mathbf{p},\mathbf{q}) = \sqrt{(p_1-q_1)^2+(p_2-q_2)^2+\cdots+(p_n-q_n)^2} = \sqrt{\sum_{i=1}^{n}(p_i-q_i)^2}.$$

The Manhattan Distance is defined as follows:

$$d_1(\mathbf{p},\mathbf{q}) = \|\mathbf{p}-\mathbf{q}\|_1 = \sum_{i=1}^{n}|p_i-q_i|,$$

In mathematics, the Euclidean distance or Euclidean metric is the "ordinary" distance between two points that one would measure with a ruler, and is given by the Pythagorean formula. The Manhattan Distance name alludes to the grid layout of most streets on the island of Manhattan, which causes the shortest path a car could take between two points in the borough to have length equal to the points' distance in taxicab geometry.

In probabilistic algorithms, Entropy (as defined in Chapter XXX) is often used as a metric of homogenity inside a distribution. Derived from that, mutual information is another one that is often used to find out how dependent two random variables *X* and *Y* are on each other.

$$I(X;Y) = \sum_{y\in Y}\sum_{x\in X} p(x,y)\log\left(\frac{p(x,y)}{p_1(x)\,p_2(y)}\right),$$

where $p(x,y)$ is the joint probability distribution function of *X* and *Y*, and $p_1(x)$ and $p_2(y)$ are the marginal probability distribution functions of *X* and *Y* respectively. Other metrics include KL-Divergence, the Bayesian Information Criterion (see Chapter XXX compression), and others. Again, we recommend studying the literature and related work around the particular problem.

Sometimes, distance metrics get invented or rediscovered around an idea because the current ones do not work as well for a specific problem.

Error Measurement and Evaluation

As explained earlier in the chapter and described in Figure 1, the error of a machine learning approach is quantified by comparing the output of the algorithms or systems against ground truth, usually human-annotations. Over the years, different error metrics have established themselves in the multimedia research community. This section will explain some of them. Just like similarity metrics, classification and clustering methods, error metrics should be chosen wisely based on the task that is to be accomplished. A wrongly chosen error metric might lead to a system being optimized towards a wrong goal. Some systems can use standard error metrics, like the following, others require specialized metrics.

The easiest way to measure error, is to classify the output labels into wrong labels and correct labels based on ground truth. The number of wrong labels is then divided by the number of total labels. The quotient can may then be expressed as a percentage. This is a very often and in many cases valid error metric. However, when a more specific distinction than wrong and false is needed, this metric may fail short of being intuitive. Often, several numbers are then used.

Another widely used statistical error metric is precision/recall. It is often used in multimedia information retrieval. Precision is the number of relevant documents retrieved by a search divided by the total number of documents retrieved by that search. Recall is defined as the number of relevant documents retrieved by a search divided by the total number of existing relevant documents (which should have been retrieved). Formally:

$$\text{precision} = \frac{|\{\text{relevant documents}\} \cap \{\text{retrieved documents}\}|}{|\{\text{retrieved documents}\}|}$$

$$\text{recall} = \frac{|\{\text{relevant documents}\} \cap \{\text{retrieved documents}\}|}{|\{\text{relevant documents}\}|}$$

Figure 8 depicts the idea.

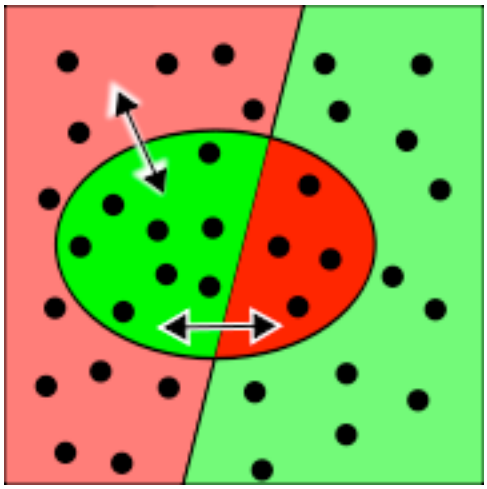

Figure 8. Recall/Precision and F-Measure depend on the outcome of a query in relation to all relevant documents and the non-relevant documents. The oval depicts the outcome of the query, the correct results are green, the wrong results are red. In this picture from [cite Wikipedia], precision is visualized by the horizontal arrow and recall by the diagonal arrow.

Outside information retrieval, the notion is usually used in a generalized manner: Precision for a class c is the *number* of true positives (i.e. the number of items correctly labeled) divided by the total number of elements labeled as belonging to the class (i.e. the sum of true positives and false positives, which are items incorrectly labeled as belonging to the class). Recall is defined as the number of true positives divided by the total number of elements that actually belong to the class (i.e. the sum of true positives and false negatives, which are items which were not labeled as belonging to the positive class but should have been). Formally:

$$\text{Precision} = \frac{tp}{tp + fp}$$
$$\text{Recall} = \frac{tp}{tp + fn},$$

with tp, fp, tn, fn, denoting true positive, false positive, true negative, false negative, respectively. A precision 1.0 for a class C means that every item labeled as belonging to class C does indeed belong to class C (but does not indicate anything about the number of items from class C that were not labeled correctly). A recall of 1.0 means every item from class C was labeled as belonging to class C (but indicates nothing about how many other items were incorrectly also labeled as belonging to class C). Therefore, the two numbers give a better indication of how the problems is attacked by a certain algorithm. Sometimes, for example, one might want to find only certain items but we should be sure that the answer is correct. This means, we want high precision low recall. When we want many positive results but don't care about them all being correct, high recall and low precision is the goal. Of course, high precision and high recall are almost always desirable in theory but usually hard to achieve in practice. Plotting the fraction of true positives vs. the fraction of false positives is called receiver-operator-characteristics or ROC curve. The so-called Equal-Error-Rate is the point at which both true positive and false positive

errors are equal. In general, the algorithm with the lowest EER is most accurate. However, often it might be desirable to choose a different operation point based on precision and recall that is more suitable for the application. For some problems and for comparative reasons, a single number, instead of a curve, might be desirable. Therefore researchers created the so-called F-score that combines precision and recall using the harmonic mean:

$$F = 2 \cdot \frac{\mathrm{precision} \cdot \mathrm{recall}}{\mathrm{precision} + \mathrm{recall}}.$$

The F-score has often been criticized as counterintuitive because too much information is abstracted away into a single value when in fact sometimes, only the histogram of the classes are a real indication of the error behavior of an algorithm.

As a final example for an application-specific error metric, we present the Word Error Rate (WER) that measures the accuracy of a speech recognition system. The general difficulty of measuring the performance of a speech recognition system is that the recognized word sequence can have a different length from the ground truth word sequence.

The problem is approximated by first aligning the recognized word sequence with the reference (spoken) word sequence using dynamic string alignment. Word error rate can then be computed as:

$$WER = \frac{S + D + I}{N}$$

with *S* being the number of substitutions, *D* is the number of the deletions, *I* is the number of the insertions, *N* is the number of words in the reference. One problem with using a simple formula such as the one above, however, is that no account is taken of the effect that different types of errors may have on the human perception and intelligibility of the result, e.g. some errors may be more disruptive than others and some may be corrected more easily than others. As explained earlier, this is a general problem with error metrics and optimizing exclusively by reducing a certain number.

Literature

- Donald Knuth in vol. 3 of The Art of Computer Programming (1973).
- Richard O. Duda, Peter E. Hart and David G. Stork: “Pattern Classification (2nd ed.)”, Wiley Interscience 680 pages ISBN: 0-471-05669-3
- Stuart J Russell, Peter Norvig: “Artificial Intelligence -- A Modern Approach”, Third Edition, PRENTICE HALL 2009.
- Raul Rojas: “Neural Networks A Systematic Introduction”, Springer 1996.
- IH Witten, E Frank: “Data Mining: Practical machine learning tools and techniques”, Morgan Kaufman, 2005.

Research Articles

- When Is "Nearest Neighbor" Meaningful? (1999), by Kevin Beyer, Jonathan Goldstein, Raghu Ramakrishnan, Uri Shaft, In Int. Conf. on Database Theory.

- John C. Platt: “Fast training of support vector machines using sequential minimal optimization”, Advances in kernel methods: support vector learning, pp 185 - 208, MIT Press, 1999
- Rosenblatt, Frank (1958), The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain, Cornell Aeronautical Laboratory, Psychological Review, v65, No. 6, pp. 386-408.
- Minsky M. L. and Papert S. A. 1969. Perceptrons. Cambridge, MA: MIT Press.
- Freund, Y. and Schapiro, R. E. 1998. Large margin classification using the perceptron algorithm. In Proceedings of the 11th Annual Conference on Computational Learning Theory (COLT' 98). ACM Press.
- Widrow, B., Lehr, M.A., "30 years of Adaptive Neural Networks: Perceptron, Madaline, and Backpropagation," Proc. IEEE, vol 78, no 9, pp. 1415-1442, (1990).
- Arthur Earl Bryson, Yu-Chi Ho (1969). Applied optimal control: optimization, estimation, and control. Blaisdell Publishing Company or Xerox College Publishing. pp. 481.
- MacQueen, J. B. (1967). "Some Methods for classification and Analysis of Multivariate Observations". 1. Proceedings of 5th Berkeley Symposium on Mathematical Statistics and Probability. University of California Press. pp. 281–297.
- Everitt, B.S. and Hand D.J. "Finite mixture distributions", Chapman & Hall (1981)
- Quinlan, J. R. C4.5: Programs for Machine Learning. Morgan Kaufmann Publishers, 1993.
- Quinlan, J. R. 1986. Induction of Decision Trees. Mach. Learn. 1, 1 (Mar. 1986), 81-106.
- Schwarz, Gideon E. (1978). "Estimating the dimension of a model". Annals of Statistics 6 (2): 461–464.
- Kullback, S.; Leibler, R.A. (1951). "On Information and Sufficiency". Annals of Mathematical Statistics 22 (1): 79–86.
- L. E. Baum, T. Petrie, G. Soules, and N. Weiss, "A maximization technique occurring in the statistical analysis of probabilistic functions of Markov chains", Ann. Math. Statist., vol. 41, no. 1, pp. 164–171, 1970.
- Viterbi AJ (April 1967). "Error bounds for convolutional codes and an asymptotically optimum decoding algorithm". IEEE Transactions on Information Theory 13 (2): 260–269
- Ward, Joe H. (1963). "Hierarchical Grouping to Optimize an Objective Function". Journal of the American Statistical Association 58 (301): 236–244.
- Hunt, M.J., 1990: Figures of Merit for Assessing Connected Word Recognisers (Speech Communication, 9, 1990, pp 239-336)
- van Rijsbergen, C. J. (1979). Information Retrieval (2nd ed.). Butterworth.

Exercises

1. Describe how the GMM implementation presented above needs to change for supervized learning.
2. Discuss usability implications for applications that use machine learning based on supervized vs unsupervized training.
3. Implement the backpropagation algorithm and use it train an MLP to learn the XOR function.
4. Use k-NN and GMMs to classify digits (see mm-creole.org)
5. Provide example multimedia tasks of when you think k-NN, GMMs, ANNs, and HMMs might provide good results. Why do you think so? Can you test it?

6. Assuming k-NN gives excellent classification accuracy. Does it still make sense to use other techniques? Why (not)?
7. So-called confidence values provide a means to estimate how certain a classifier is of a decision. This is often desirable when integrating different media together so the different decision can be weighted against each other. Discuss ideas on how to come up with confidence values.
8. Explain strategies for the combination of different media based on GMMs, ANNs and Decision Trees.
9. Explain visually, how an HMM can help in speech recognition.
10. Provide four examples for applications that require low/high precision/recall (respectively)
11. Provide two examples where Word Error Rate does not reflect perceptual error.
12. Explain what happens when models overfit or underrepresent. Discuss strategies to avoid these problems.